Redirecting by Injector

Robert E. Filman
RIACS
NASA Ames Research Center, MS 269/2
Moffett Field, CA 94035
rfilman@mail.arc.nasa.gov

Diana D. Lee
SAIC
NASA Ames Research Center, MS 269/2
Moffett Field, CA 94035
ddlee@mail.arc.nasa.gov

Abstract

We describe the Object Infrastructure Framework, a system that seeks to simplify the creation of distributed applications by injecting behavior on the communication paths between components. We touch on some of the ilities and services that can be achieved with injector technology, and then focus on the uses of redirecting injectors, injectors that take requests directed at a particular server and generate requests directed at others. We close by noting that OIF is an Aspect-Oriented Programming system, and comparing OIF to related work.

1. Introduction

Traditional software system development is a monolithic process. An organization building a software system was presumed to know how it wanted that system to behave. The requirements for that behavior would flow down to the construction of the underlying modules. Since the modules were being built specifically for the system in question, it was "straightforward" to get their developers to obey rules and conform to defined standards. To the extent that the system used an externally provided component such as a GUI or database, the behavior of that component could be ascertained and the use of that component within the system shaped to match the external component's actual behavior.

Software development has gotten more complex. Technologies such as CORBA and HTTP provide the glue for building applications from distributed components. But understanding the nuances of multiple components and varieties of glue is itself an intellectual challenge. We can't expect an application programmer, seeped in knowledge of the application domain, to also become expert in the intricacies of many components, even if the application needs to use them all. Similarly, components impose their own constraints on their usage. We want to develop systems from components but don't want the artifacts of a particular component manufacturer to permeate our designs, rendering us eternally dependent on the whims, demands and destiny of that vendor. We want components that obey our policies—not to have to distort our systems to match the policies of the components. And we want ways to federate existing systems while still maintaining overarching rules and procedures.

Distributed systems introduce additional complexity beyond simple componentization. Developing a distributed system is more difficult because distributed systems are concurrent and nondeterministic, because distribution introduces many additional kinds of failures, because distribution is naturally less secure, and because distribution's inherent decentralization is inconvenient to manage. Distributed computing can be made simpler by making it look more like conventional programming and by providing and automatically invoking correct implementations of distributed and concurrent algorithms.

To deal with some of the difficulties of creating and evolving distributed systems, we created the Object Infrastructure Framework (OIF). We wanted ways to achieve "ilities" such as reliability, security, manageability and quality of service without burdening application programmers with the details of knowing how these ilities are programmed and when to apply the ility action.

2. The Object Infrastructure Framework

The primary mechanism used in OIF is *injecting* behavior on the communication path between components. This effectively serves to wrap services with additional actions at both the client and server ends. The following features distinguish the OIF wrapping mechanism:

- **Discrete injectors.** Our communication interceptors are discrete, first class objects. Therefore, they have (object) identity and can be sequenced, combined and treated uniformly by utilities.
- **Paired injectors.** An ility may require injecting behavior on both the client and server of a distributed system. For instance, authentication can be implemented by having a server-side injector check the credentials generated by a client-side injector.
- **Injection by object/method.** Each instance and each

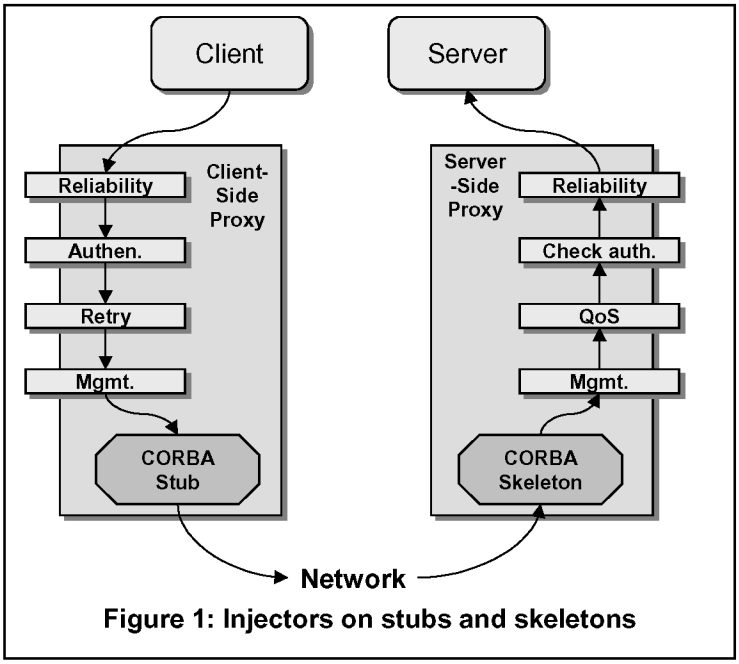

Figure 1: Injectors on stubs and skeletons

method on that object can have a distinct sequence of injectors.

- **Dynamic injection.** The injectors on a stub can be changed during the execution of a system, allowing, for example, the placement of debugging and monitoring probes or the replacement of old versions of software with newer ones.
- **Annotations**. Annotations on requests and responses provide a channel for inter-injector communications. These annotations are name-value pairs. Injectors are capable of reading and modifying annotations (and reading and modifying the request arguments and target function name).
- **Thread contexts.** Our goal is to keep the injection mechanism invisible to the functional components. However, sometimes clients and servers need to communicate with injectors. We make annotations largely transparent to functional components by providing an alternative communication channel. Each client and server thread has annotations, its *thread context*. The system arranges to copy annotations among the client's thread context, the request, and the server's thread context.
- **High-level specification language and compiler.** To span the gap between abstract ilities and discrete sequences of injectors we created a compiler, Pragma, that takes a high-level specification of desired properties and ways to achieve these properties and maps that specification to an appropriate set of injector initializations.

Figure 1 illustrates the relationship of injectors to CORBA skeletons and stubs. A more complete discussion of the motivation for these features can be found in reference [5].

3. Injectors

OIF injectors work with CORBA stubs and skeletons that have been modified to obtain the injector sequence for each method and to invoke the first injector in that sequence with (1) a (classical CORBA) request object that includes (a) the target server, (b) the operation to be performed on that server, (c) the arguments of that operation, and (d) a set of annotations for this operation, and (2) the *continuation:* the set of injectors to be executed after this injector. Annotations are name-value pairs, where the name is a string and the value, any CORBA value. The injector can modify the target, the operation arguments, the annotations, and the return value. It can also invoke arbitrary other remote calls, and can itself be a CORBA-visible object, capable of handling service requests from other sites.

Grossly, an injector wants to perform some actions *before* the server action and some *after*. It is the responsibility of an injector to invoke the remaining injectors of the continuation between its before and after actions (that is, to call the "next" operation on the continuation.) This structure allow injectors to alter the flow of control in interesting ways—for example, to forgo calling the after injectors (as is done in the caching injector, which uses its cache of prior service calls values to avoid redundant calls) and to use the natural exception-catching mechanisms to catch (and correct) exceptions in the continuation processing.

Injectors can be used for achieving ilities such as reliability, security, manageability and quality of service, and can also be profitably employed in improving the computational efficiency of distributed systems. Table 1, from [7] list some applications of injectors.

Space limitations preclude a detailed description of OIF's implementation. Briefly, OIF has an alternative IDL compiler whose proxies include calls to the proxy-specific sequence of injectors. An injector maintains the request object/annotation/thread-context relationships. Pragma works by creating initialization tables for the mapping from interface classes and methods on those interfaces to the sequence of injector factories to be invoked in creating a stub. Figure 2 illustrates the process and structure of building an OIF application. This figure shows that the application IDL and the OIF Pragma specification are run through the IDL and Pragma compilers, creating code that is linked with the application code and elements of injector libraries to make the complete application. Reference [4] has more detail on these mechanisms.

4. Redirecting injectors

A client-side injector can change the destination of a request, or use the occasion of a request as a reason for generating requests to other objects. This suggests several possibilities

- A *rebind* injector can catch failures on the part of the original target and redirect the request to another target that offers the same service. This process can be repeated through the set of alternative service providers known to the injector.
- An *impatient* injector, knowing of several targets that offer the same service, can simultaneous send the same request to them all of them. (Of course, if every object was impatient, performance would suffer. Ironically, under straightforward charging policies for priority service, the appropriate local behavior is to be impatient at low priority [2].)
- An *insecure* injector sends the same request to several targets offering a service and combines their responses. For example, such an injector might average numeric values, take a majority vote or infer the intent of a service on the basis of its perforation.
- A *mediating* injector partitions the problem into subproblems, sends the requests for different parts of the subproblems to appropriate targets and combines their answers back into a whole.
- A *balancing* injector knows several targets that offer the same service and sends the request to one with the hope of balancing the overall system load. This decision might be based on a random selection from the possible targets, on a learning algorithm working off the injec-

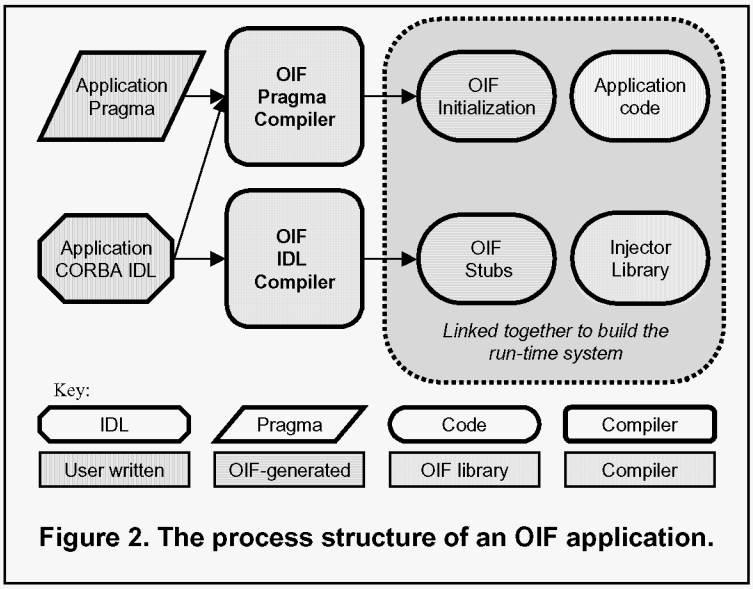

Figure 2. The process structure of an OIF application.

Table 1. Injector applications

Ility	Injector	Action
Security	Authentication	Determines the identity of a user.
	Access control	Decides if a user has the privileges for a specific operation.
	Encryption	Encodes messages between correspondents.
	Intrusion detection	Recognizes attacks on the system.
Reliability	Replication	Replicates a database.
	Error retry	Catches network timeouts and repeats call.
	Rebind	Notices broken connections and opens connections to alternative servers.
	Voting	Transmits the same request to multiple servers (in sequence or parallel) combining the results by temporal or majority criteria.
	Transactions	Coordinates the behavior of multiple servers to all commit or fail together. Requires additional interface on application objects.
Quality of service	Queue-manager	Provides priority-based service.
	Side-door	Provides socket-based communication transparently to application.
	Futures	Provides futures transparently to the application.
	Caching	Caches results of invariant services.
Manageability	Logging	Reports dynamically on system behavior.
	Accounting	Reports to accounting system on incurred costs.
	Status	Accrues status information and reports when requested.
	Configuration management	Dynamically test for incompatible versions and automatically updates software.

tor's historical experience with the targets, or on an externally provided "traffic report" on the load on various targets. The balancing injector might itself become a source of traffic information to its correspondents [8]. [Clearly, traffic information is of more use for requests that take a long time (e.g., compute this computational fluid dynamics problem), than for ones that can be handled quickly (e.g., what's the price of this stock right now.)]

The notion of a redirecting injector raises the question, "How does the redirecting injector know where to redirect?" The question is not as trivial as it sounds. There are four reasons we may not be able to get the redirection information from the target object:

- The target may not possess an interface for such questions. The provider of a stock-quote service built a component to provide stock quotes, not to provide a list of providers of stock quotes.
- The alternative services may be competitors. That is, the stock quote provider is likely (for whatever economic reason) to want to be the one providing the client with stock quotes.
- The target may not even know of the existence of the alternative services, or even understand that a particular service, with the right mediation, can be used as a substitute for its computations.
- The reason we're often seeking an alternative provider is often precisely because the original target has failed, making it a poor candidate for providing advice.

We note two additional complications.

- References to objects can arrive through complex compositions of method call arguments and return values. We do not necessarily obtain a reference to an object directly from that object itself.
- In general, we want to say more about an object than just alternatives that provide the same service—we may also want information about an object's accuracy, reliability, security problems, congestion, and so forth. This *commentary* may be generated dynamically as the application runs, and is likely to come from other components "sharing" their experiences. We ought to expand any mechanism that works for redirection information to these other forms of commentary.

In the following discussion, we define a *clerk* as a component that is a database of commentary. That is, we imagine some calls to a clerk are of the form: "Assert a property P of component X is y," and others are "Query what is (or are) the values of the P property of X?"

Possibilities for organizing the sharing of commentary include

- A component in need of commentary about an object could appeal to a *famous* (globally well-known) clerk. In some sense, directory and search engines such as Yahoo and Google serve this purpose for the Internet as a whole; imdb.com (the Internet Movie DataBase) and deja.com (Usenet) are repositories of user commentary on particular topics. Famous clerks have the advantage that they can be programmed as "constants" into applications and that the keepers of these clerks are likely to keep them running. They have the disadvantage of making public all shared information and of focusing (by providing easiest access) on information the clerks deem interesting.
- The developer of an application could set up one or more application-specific clerks. Components created in that application could know, from their initiation or other methods, of the existence of these clerks. This has the advantage of being a straightforward solution for tightly integrated applications, but the disadvantage of demanding common knowledge among the components of a loosely-coupled application.
- A component could keep track of the components with which it had communicated (its *acquaintances*). Needing information about an object, the component could query its acquaintances recursively until one was found with the necessary information. After all, if we've come to know of an external object, it must be because one of our acquaintances told us about it. (This querying could be done either in a distributed fashion, by marking the query message with a unique symbol and having queried acquaintances not propagate messages they had seen before, or in a centralized fashion, where the query for information returned either the information or the set of acquaintances who might have the information, to be poked again by the original inquirer.) More clever implementations of these algorithms might cache information such as the answers to commonly asked questions (e.g., "Do you have an alternative server for A?" and "Who are your acquaintances?") The disadvantage of this approach is that it can imply a considerable amount of dynamic work on raising a question—we would be actively and unboundedly searching the network, and might also require a considerable amount of intermediate storage to keep track of acquaintances, cached answers and recent questions.
- Better than relying on famous clerks, the application might arrange dynamic clerks. That is, component creation would require the creation, assignment or location of a local clerk for that component for each variety of commentary. Injectors on components would communicate their clerk as part of the annotations of their ordinary requests; clerks would be informed of the discovery of new, previously unknown clerks, and the clerks themselves made responsible for organizing themselves, keeping commentary about components, and answering questions about this commentary.

We are currently implementing redirecting injectors, and have allowed ourselves to be distracted into exploring this last alternative. Clearly, clerks could search among themselves for commentary they lack. The problem is not

as bad as such search at the component level, as there are likely to be far fewer clerks than components, but the prospect of doing dynamic, unbounded search is unsettling. As an alternative, we are currently implementing the *Captain* algorithm. Clerks associate information with objects. A *community* of clerks, where each member of the community knows of all the others, can partition the commentary about all objects among themselves, relying on a hashing or b-tree algorithm to quickly determine which community member stores the information about a specific component. When two otherwise disjoint communities learn about each other, they need to reorganize their collective information. This reorganization happens under the supervision of one of the community's captains. Complexity arises in this algorithm if while two communities are merging, another community is discovered. To handle such situations, Captain does the reorganization transactionally.

5. Aspect-oriented programming

We have described a mechanism for separately specifying system-wide concerns in a component-based programming system and then weaving the code handling those concerns into a working application. This is the theme of Aspect-Oriented Programming (AOP). OIF is an instance of AOP, and brings to AOP a particularly elegant division of responsibilities. Key work on AOP includes Harrison and Ossher's work on Subject-Oriented Programming [8] which extends OOP to handle different subjective perspectives; the work of Aksit and Tekinerdogan on message filters [1], that reifies communication interceptors; Lieberherr's work on Adaptive programming [12] that proposed writing traversal strategies against partial specifications; and Kiczales and Lopes [10] work on languages for separate specifications of aspects, which effectively performs mixins at the source-code language level.

In reference [6], we argued that the two primary mechanisms for implementing AOP systems are "clear-box" approaches, where a compiler or interpreter examines the source of the application and can arbitrarily manipulate that source, and "black-box" (or wrapping) approaches, where the aspect mechanism is arranged as a layer around the component, achieving aspects by manipulating what goes in and out of the component. Like message filters [1], Aspect Moderator [2] and Synchronization Rings [10], OIF seeks AOP by wrapping.

The idea of intercepting communications is not new to AOP. Perhaps the earliest examples were in Lisp: the Interlisp advice mechanism and mix-ins of MacLisp. A more modern realization is seen in mediators [14], which recognizes the implicit agent-hood of the communication interception elements.

More recently, the CORBA standard has been extended to provide *interceptors*, programmer-defined operations that run in the communication path. From our point of view, this is the right idea, wrongly implemented. CORBA interceptors run after the call's arguments have been marshaled, making them opaque to the interceptor code (though well-situated for encryption). CORBA interceptors are also considerably more structurally rigid than the OIF framework's injectors, not being objects to be manipulated at run-time. If one is particularly fond of CORBA interceptors, one can view our work as a methodology for using them.

Thompson *et. al* [13] present an OIF-like use of injector-like plug-ins in a web architecture. Examples of uses of these plug-ins include performance monitoring and collaborative documents.

It is common to tackle ility concerns by providing a framework with specific choices about those concerns. Examples of such include transaction monitors (e.g., Encina, Tuxedo) and distributed frameworks like Enterprise Java Beans and CORBA.

6. Concluding remarks

We have taken an aspect-oriented approach to injecting reliability into a distributed system. Our black-box approach lends itself readily to integrating components whose source code may not be available. Furthermore, because black-box techniques do not depend on particular implementation of components, the result is generally more reusable and maintainable than clear-box methods. We have demonstrated the ideas of OIF in the context of CORBA distributed systems. However, the basic ideas of intercepting communications (wrapping), annotating requests, reifying interceptors, dynamically choosing which intercepts to run and providing high-level specification mechanisms to mapping injector requirements to code can be applied to any other environment where a wrapping can be imposed on program elements.

We have successfully demonstrated reliability in the form of error recover, redundancy, mediation, and load balancing; we have also demonstrated these mechanisms are extensible to other concerns such as security, quality of service, and manageability.

In studying reliability, we have come to understand that there is a special complexity of finding alternate servers. One of the goals of this paper has been to broach the question of arranging for decentralized shared knowledge in peer-based communicating systems. We have proposed one model for such shared knowledge that relies on imposing additional structure into remote calls but suggests no other shared information. In OIF, this imposition is straightforwardly arranged by the addition of an injector. Other approaches are possible. For example, a system generating its own communication mechanism could build such tracability into its natural structure. It remains to be seen whether such zero-common-knowledge shared anno-

tation will prove important to some class of future applications.

7. Acknowledgments

Our thanks to Tarang Patel and Alex Shaykevich for their comments on the drafts of this paper.

8. References

[1] M. Aksit and B. Tekinerdogan, "Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters", In S. Demeyer and J. Bosch (Eds.), *Object-Oriented Technology ECOOP'98 Workshop Reader,* Springer-Verlag, Berlin, 1998 http://wwwtrese.cs.utwente.nl/Docs/Tresepapers/FilterAspects.html

[2] A. Bader, C.A. Constantinides, T. Elrad, T. Fuller, and P. Netinant, "Building Reusable Concurrent Software Systems", *Proc. Int'l Conf. on Parallel and Distributed Processing Techniques and Applications,* CSREA Press, Las Vegas, 2000, Vol II, pp. 845-851.

[3] A.M. Bell, W.A. Sethares, D.L. Reiley, D. Wolpert, K. Tumer, and J. Frank, "Strategic Behavior, Learning and the Efficient Allocation of Network Resources", *The International Congress on Networks, Groups and Coalitions*, Manresa, Spain, May 1999.

[4] R. Filman, "A Software Architecture for Intelligent Synthesis Environments", *Proc. 2001 IEEE Aerospace Conference*, Big Sky, Montana, March 2001.

[5] R. Filman, S. Barrett, D. Lee, and T. Linden, "Inserting Ilities by Controlling Communications", *Communications of the ACM*, in press. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/oif-cacm-final.pdf

[6] R.E. Filman and D.P. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness", *Workshop on Advanced Separation of Concerns*, OOPSLA 2000, October 2000, Minneapolis. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/aop-is.pdf

[7] R.E. Filman, D.J. Korsmeyer, and D.D. Lee, "A CORBA Extension for Intelligent Software Environments," *Advances in Engineering Software 31* (8-9), 2000, pp. 727-732 http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/williamsburg-print-final.pdf

[8] R. Filman, and T. Linden, "SafeBots: A Paradigm for Software Security Controls," *Proc. ACM New Security Paradigms Workshop*, Lake Arrowhead, CA, September 1996, pp. 45–51.

[9] W. Harrison, and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", Proc. OOPSLA '93. *ACM SIGPLAN Notices 28* (10) 1993, 411-428.

[10] Holmes D., Noble J. and Potter J., "Towards Reusable Synchronisation for Object-Oriented Languages", Aspect-Oriented Programming Workshop, ECOOP'98, 1998 http://www.mri.mq.edu.au/~dholmes/research/aop-workshop-ecoop98.pdf

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming", In *Proc. ECOOP '97, LNCS 1241,*: Springer-Verlag, Berlin, 1997, pp. 220-242. http://www.parc.xerox.com/ spl/projects/aop/tr-aop.htm

[12] K. Lieberherr, *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, 1996.

[13] C. Thompson, P. Pazandak, V. Vasudevan , F. Manola, M. Palmer, G. Hansen, and T. Bannon, "Intermediary Architecture: Interposing Middleware Object Services between Web Client and Server", *Computing Surveys*, in press. http://www.objs.com/OSA/Intermediary-Architecture-Computing-Surveys.html

[14] G. Wiederhold, "Mediators in the Architecture of Future Information Systems", *IEEE Computer 25*, 1992, pp. 38-49.